

CS261 Lecture 6 (9/16)

General Sandboxing:

We have some host program, that runs a bit of untrusted code inside the sandbox, the guest code. The host, typical application is your web browser, want to run some flash or web plugin. The host wants to let the plugin run in the browser's memory space, but don't entirely trust the plugin. Two options to implement this:

1. Make all pages read only. Then let the code run. Let plugin use api. Switch page table around. The calls are very expensive, like a context switch.
2. Make domain call very fast. So use masking to make sure it stays within the region it's suppose to be.

PittSField:

PittSField use masking to limit where it can write to b/c...

- Software needs writes.
- User wants to run code needs more access, so it will call the OS, which will cause a context switch and increase overhead.

PittSField does **NOT** defend against **buffer overrun** vulnerabilities. The goal of PittSField is sandboxing, not preventing buffer overrun. It wants to run non- malicious guest code and contain malicious ones. Given a piece of guest code, PittSField wants to make sure it won't be able to attack the host.

Fast – special memory layout. Can write to any address when couple of the first four bits are cleared.

Assumption – any jump can go to any basic block, which introduce risk that can jump to anywhere that bypasses the check. If we want proof, we'll fall short of that because the check is in another basic block. It's harder to do in the binary world. In the code world, we have control flow graph, so can do static analysis.

Weaknesses of papers:

- There was no performance info on comparison with cross domain calls. Often measure within domain call, not cross domain call.
- Did not compare against it using a separate process.

Byte Level Taint Tracking:

Suppose we want byte level taint tracking, but don't have C source code, only binary. How could you do it? Could you do it? How might you try to introduce it. I want to take binary and do taint tracking and ensure that control transfer, the branch target is never tainted. If it ever jumps to a taint address, I want to stop execution.

- Dynamically rewrite to go to fix location (support structure), then check if tainted, then go to the right place if it's ok.
- Segment registers – have data segment disjoint code segment. All jumps to code segment, not data segment. But still want it to not go to code segment if code segment if tainted.

- Keep taint vector to see what bytes (bit for every byte - memory) are tainted. Keep taint status of registers and memory. For register – reserve one register that will hold the taint status of all other registers (won't work if you use %esx, and compiler also uses %esx, and you don't have control over the compiler). So spill compiler need register into memory, read in memory data to register. Access to %ecx will be a sequence of read and writes. Make it more efficient by seeing which registers are alive or dead. Use dead ones to load in values to memory
- Multiple translations – have diff translation of the basic block. Have different combinations of the taint. Have to maintain taint record of all registers in memory.

Multi-threaded program has problem if taint information writing and reading is not atomic. There will be race conditions.

Baggy Bounds Checking:

Want to instrument legacy code to have baggy bounds checking. It is a pain to do it on both stack and heap. So if you only want bounds checking for heap allocated objects and treat heap as one giant object, the software will still vulnerable to stack smashing. Problem is that there is no type information so we won't know which one is pointer and which one is integer.

- Relies on static optimizations to remove the check.
- To improve efficiency of loops, put the check outside of the loop. Example:

```
for (int l = 0; l < n; l++){
    Check(p); // (n checks, inefficient)
    *p++ = l;
}
```

Instead do this:

```
Check'(p, n) // (check that this is in the same inbound object as p) (1 check only!)
For (int i = 0; i < n; i++){
    *p++ = l;
}
```

Web browser scenario:

PittSFeld paper told us how to isolate the guest code to run outside sandbox. If it's plugin code, we want it to be able to communicate with browser, like an api or somewhere it can jump to in the browser code. What would we have to do to enable plugin to have a cross domain call?

- Have some stub code (written by browser) that can jump outside the sandbox. Stub is not isolated or translated with PittSFeld in any way. Changes you have to make:
 - o Allow it to jump to that address in the fixed address in stub code. Allow the jump w/o masking, like an exception.
- What if plugin does something you don't want it to do, then does the stub code. Add stub code to make sure the return address is ok. Stub code was passed in a pointer that the stub code will write to. If it writes to the return address.

Different ways to implement the OS, security level it provides differences

MMU-based	SFI-based
<ul style="list-style-type: none">• No translation is need,• Hardward performance overhead• more flexible.• Flush caches when context switch• Availability – preemptive time scheduling.	<ul style="list-style-type: none">• Risk of bugs in translation• Better performance b/c no need to flush caches when context switch• Privacy is harder to guarantee under SFI, b/c it may not restrict reads• Availability – timer interrupt.

SFI in the Real World

PittSFIeld is not actually used because it's just a research prototype. It assumes assembly code is available. It doesn't work with legacy code, has 30% overhead.

VMware – dynamic binary to binary. on the fly binary translation of basic block. Very light check. For every indirect branch, then use vs32 technique. Look at hashtable, if not there, translate.

NaCl (google) – assembly/source -> binary. download x86 code from the web and run it in browser close to full speed. Morph of vx32 and PittSFIeld. You have source code and want to compile to .exe that can embed to webpage. Have you compile your code of gcc. It ensures 32 byte alignments. Inserts segmented address space. One region starts at 0 for data. Another address 0 for code. Insert explicit segment overrides. Use code and data segment registers for writes. Use PittSFIeld mask technique and code segment register for jumps. Run time overhead of about 5%.